Debreceni Egyetem

Informatikai Kar

Sorbanállási rendszerek szimulációja

Témavezető:

Dr. Sztrik János

egyetemi tanár, az MTA doktora

Készítette:

Ficsor János

mérnök informatikus BSc.

Debrecen

2008.

Ezúton szeretném megköszönni témavezetőmnek, Dr. Sztrik Jánosnak a segítségét és támogatását, melyet a dolgozat, és a program elkészítéséhez nyújtott.

Tartalomjegyzék

1. Bevezetés

A sorbanállás elmélet az a tudomány, amely olyan rendszereket modellez, amikben igények sorokat kialakítva várakoznak egy vagy több erőforrásra. Ezekkel a modellekkel pedig elemzéseket készíthetünk a rendszer működéséről, és tulajdonságairól. Előfordul viszont, hogy egy valós rendszerhez olyan bonyolult matematikai modell tartozik, amivel nagyon nehéz, és körülményes a rendszer jellemzőinek meghatározása. Olyan eshetőség is létezik, hogy az adott rendszert nem lehet matematikai modellel leírni. Ilyenkor jönnek jól az olyan eszközök, mint amik egy szimuláció során mért adatokból a statisztika eszközeivel határozzák meg a rendszer tulajdonságait.

Mivel a sorbanállás elmélet meglehetősen fiatal, így az oktatásban sincs még jelentősebb hagyománya. Tapasztalataim alapján ezt a tudományt megismerni vágyó hallgatók az előadásokon leginkább a matematikai modellezéssel ismerkednek meg. Kihívást jelenthet viszont a matematikában kevésbé járatos emberek számára a tanagyag megértése, így egy olyan segédeszköz, ami vizualizálná a képletek sokaságát, nagyban segíthetné az anyag elsajátítását, megértését. Olyan módon is fel lehetne használni ezt a segédeszközt, hogy a matematikai modell által szolgáltatott eredmények összevethetőek lennének egy szimulációs kimenettel, így könnyen ellenőrizhetővé válnának az órai feladatok megoldásai, illetve lehetőség nyílna arra is, hogy új témaként felmerülhessen a szimuláció és modellezés közti különbség, eltérés megvitatása.

Maradva az oktatásnál, egy olyan program, ami kézzelfoghatóvá, és látványossá tenné a sorbanállás elméletet, akár népszerűsíteni is tudná ezt a tudományt. Felkeltve olyanok érdeklődését, akik esetleg még nem hallottak róla. Így növelni lehetne azoknak az embereknek a számát, akik a későbbiekben komolyabban akarnak foglakozni a témával, és ezzel segítenék eme tudomány fejlődését.

Manapság a számítástechnika olyan szinten van, hogy egy több ezer eseményből álló szimulációs folyamat sem terheli meg komolyan a számítógép erőforrásait – ez alatt azt értem, hogy pár másodperc alatt előáll a kimenet. Szimulációs modelleket sokféle képen lehet implementálni. Használhatóak az általános célú programnyelvek, mint például a C, C++, vagy a Java, vagy speciális szimulációs nyelvek, mint a GPSS, SIMAN, vagy a SIMSCRIPT. A

programozók számára ismertebbek az általános célú programnyelvek, de nem biztos, hogy azok rendelkeznek a szimulációhoz szükséges beépített eszközökkel. Ezek nélkül egy szimulációs modell elkészítése összetett, és fárasztó feladattá válhat. A speciális szimulációs nyelveket viszont először meg kell tanulni, mielőtt a modell megvalósításába kezdenénk. Ezeknek a nyelveknek viszont közel sincs olyan széles körű támogatottságuk, és elérhetőségük, mint a legtöbb általános célú programnyelvnek.

Szakdolgozatomban szeretnék egy olyan programot bemutatni, ami a matematikai modellezés helyett szimuláció segítségével szolgáltat információt a vizsgálandó sorbanállási rendszerről. Ez a program a *Qsim*. A programot a szakdolgozat számára készítettem, megírásához a Java programozási nyelvet használtam, és az *SSJ – Stochasitc Simulation in Java* keretrendszert, amit Pierre L'Ecuyer felügyelete alatt a Montreali Egyetemen fejlesztettek ki. A *Qsim* megtalálható a dolgozathoz mellékelt CD-n. A program képes különböző sorbanállási rendszereket meghatározott ideig szimulálni. A futás végén pedig táblázatban foglalja össze a legfontosabb jellemzők értékeit. Továbbá képes grafikonon ábrázolni az egyes adatcsoportok mérési eredményeit. Így eseményről-eseményre nyomon követhető a rendszer állapota. A grafikon ábrázolja a mért adatokból számított átlagot, és a megadott százalékú konfidencia intervallumot is.

Célom egy olyan eszköz létrehozása volt, ami segítségével egyszerűen adható meg a vizsgálni kívánt sorbanállási rendszer típusa, és könnyedén értelmezhetőek a kimeneti adatok is. Így a program nem csak az előbb említett oktatási területen hasznosítható, hanem a valós világ *egyszerűbb* sorbanállási rendszeri elemzésére is. Így ha valaki optimalizálni szeretné az ilyen igényeket és erőforrásokat tartalmazó rendszerét, nincs szüksége bonyolult, összetett és drága programokat megvásárlására, nem beszélve a ráfordított időről, ami alatt a szoftver kezelését kell megtanulnia.

Szakdolgozatomban ismertetem a sorbanállás elmélet alapvető fogalmait, majd részletes leírással szolgálok a program elkészítéséről és működéséről. Programozási, és elméleti szempontból taglalom a szimuláció inicializálását, a szimulációs folyamat menetét, és a statisztikai adatok gyűjtését, rendezését. Részletesen leírom, hogy a mért adatokból hogyan kaptam a grafikonok görbéit, illetve megismertetem az egyik legnépszerűbb módszert, amivel egyensúlyi állapotban lévő rendszerek mért adataiból meghatározható az átlaghoz tartozó konfidencia intervallum. Szeretném a program helyes használatát, és lehetőségeit is

bemutatni, így a dolgozat azon része úgy is értelmezhető, mint a program kézikönyve. Végül összevetem egy ismert sorbanállási rendszer matematikai modellje által kapott rendszerjellemzők értékeit a *Qsim* szimulációja során mért értékekkel.

2. Általános sorbanállás elméleti ismeretek

Ahhoz hogy egy sorbanállási rendszert megvizsgálhassunk, ismernünk kell a jellemzőit. Tudnunk kell, hogy hány kiszolgáló áll a rendelkezésre, milyen gyakran jönnek új igények, mennyi ideig tart őket kiszolgálni. Véges-e a rendszer kapacitása, vagy végtelen? Milyen sorrendben történik a kiszolgálás?

2.1 A Kendall-féle jelölés

David G. Kendall 1953-ban bevezette az *A/B/C* jelölést, melyben az *A* az beérkezési időközök eloszlása, a *B* a kiszolgálási idők eloszlás, a *C* pedig a kiszolgálók száma.

Az *A* és *B* paraméterek megválasztására a *Qsim* a következő lehetőségeket kínálja a felhasználó számára:

- Exponenciális eloszlás (M)
- Erlang eloszlás (Er)
- Hiperexponenciális (HrE)
- Hipo-exponenciális (HoE)

Manapság egy rendszer leírásához az *A/B/m/K/N* jelölést használják, ahol az *m* az eredeti *C*-nek felel meg, még a *K* a rendszer kapacitását jelenti – azt, hogy összesen hány igény tartózkodhat a várakozási sorban, és a kiszolgálóegységeknél összesen. Az *N* az igényforrás számosságát jelöli. Meg szokták még adni a kiszolgálási elvet is, ez leggyakrabban FIFO – az elsőként érkezett igény lesz először kiszolgálva, vagy LIFO – az utolsóként érkezett igény lesz elsőként kiszolgálva. A SIRO (Service In Random Order) kiszolgálási elv véletlenszerűen választ a várakozó igények közül. Ha az igényekhez prioritást lehet hozzárendelni a fontosságuk alapján, akkor ezen a prioritáson alapul a kiszolgálás sorrendje. Ez az egyik legalkalmasabb ütemezési elv, mivel így az igények közti fontossági sorrendet felállítva történik a kiszolgálás. [1]

Vegyünk például egy egyetemi menzát: Tegyük fel, hogy a beérkezés Poisson-folyamat szerinti. Ekkor a beérkezési időközök exponenciális eloszlásúak. Legyenek a kiszolgálási idők is exponenciális eloszlásúak. Az emberek 1 sorban állnak várva, hogy valamelyik kasszánál

fizethessenek. Tegyük fel, hogy 2 kassza van. A kiszolgálási elv nyilván FIFO, mert az fizet először, aki hamarabb érkezett a menzára. A rendszer kapacitását végesnek tekinthetjük akkor, ha a sor csak a menza ajtajáig tarthat, mert ekkor már nem fér be újabb ember. Ha feltesszük, hogy 48 ember állhat maximum sorba, akkor ez egy *M/M/2/50/FIFO* rendszernek tekinthető. Viszont tekinthetjük a rendszer kapacitását végtelennek, ha az emberek nem törődnek az ajtóval, és az utcán is várakozva végtelen hosszú sort alkothatnak. Ebben az estben egy *M/M/2//FIFO* rendszert kapunk.

2.2 A sorbanállási rendszerek jellemzői

A sorbanállási rendszerek teljesítményének, és hatékonyságának vizsgálata közben a következő mérőszámokat fogjuk használni:

- az igények várakozási ideje *(az az idő, amit egy igény a sorban tölt el a kiszolgálóra várva)*
- kihasználtság *(a foglalt szerverek átlagos száma)*
- kiszolgálási idő *(az az idő, amit egy igény a kiszolgálónál tölt el)*
- válaszidő *(a várakozási idő és a kiszolgálási idő összege)*
- igényvesztés aránya *(Véges kapacitású rendszereknél előfordulhat olyan, hogy egy újonnan érkező igény már nem fér be a sorba, így az elvész. Ha ekkor a kiszolgált igények számát elosztjuk az összes érkezett igénnyel, akkor az a hányados megadja az igényvesztés arányát.)*

A szimuláció szempontjából ezeket az értékeket különböző szemszögből mérhetjük. A *várakozási időt, kiszolgálási időt*, és a *válaszidőt* az igények szempontjából vizsgáljuk. Azaz, csak akkor végzünk új statisztikai mintavételezést, ha érkezik egy igény. Így a grafikon *x*-tengelyén a beérkezett igények sorsszáma fog megjelenni.

Vannak olyan esetek is, amikor érdemes a szimuláció időtartamát felosztani, és így az időt felvenni az *x*-tengelyre. Ilyen eset a *kihasználtság* vizsgálata. Hiszen képzeljük el, ha a kihasználtságot az igény szemszögéből mérnék: Legyen a vizsgált rendszer M/M/1/1 típusú. Ez ugye azt jelenti, hogy nincs várakozási sor, egy igény vagy azonnal kiszolgálásra kerül, vagy elvész. Ha az igény szempontjából vizsgálnánk, akkor a foglalt szerverek átlagos száma mindig 0 lenne, hiszen az olyan igény elvész (és így mérés sem készül), amelyik érkezése

pillanatában telítve találja a rendszert. Az olyan igény, amely viszont üres rendszerhez érkezne, mindig azt látná, hogy a kiszolgáló tétlen, azaz a statisztikai megfigyelésének értéke szintén mindig 0 lenne. Ilyen esetekben úgy célszerű eljárni, hogy a szimulációs időt felosztjuk n darab, azonos nagyságú l időintervallumra, úgy hogy $n \cdot l = a\ szimuláció\ időtartama$. Majd a szimuláció során minden intervallum kezdetén mérést végzünk a rendszer állapotáról. Így az idő függvényében előállt grafikon már egyszerűen, és látványosan értelmezhető.

3. A *Qsim* program implementálása, és működése

A *Qsim* program elkészítésével az volt a célom, hogy létrehozzak egy olyan programot, ami szimuláció segítségével képes sorbanállási rendszerek tulajdonságait meghatározni. Ellentétben a sorbanállás elméletben leggyakrabban alkalmazott matematikai modellezéssel, az itt kapott eredmények minden szimuláció után más-más értéket vesznek fel (ugyanis a háttérben véletlen szám generátorok vannak). A program számol szórást is, és konfidencia intervallumot is.

A *Qsim* megírásához a Java programozási nyelvet használtam. Azért esett erre a választásom, mert a Sun Microsystems által kifejlesztett objektumorientált nyelv platform független, széles körben támogatott, és kiváló integrált fejlesztő eszközök állnak a programozók rendelkezésére, szintén ingyenesen. Szeretném kiemelni a *NetBeans IDE 6.1* rendszert, amit a fejlesztés folyamán használtam. A fejlesztő környezet letölthető a http://www.netbeans.org/ honlapról. Visszatérve a platformfüggetlenségre: fontosnak tartottam, hogy a programom mind Windowsos, mind Linuxos környezetben működhessen, hiszen manapság egyre elterjedtebbek a Linuxos gépek a drága, lassú, és megbízhatatlan Windowsos rendszerekkel szemben.

Mivel a Java egy általános célú programozási nyelv, így alapból nem biztosít könyvtárakat diszkrét események szimulációjára. Így a *Qsim* a SSJ – Stochasitc Simulation in Java keretrendszer osztályait használja a szimulációs folyamatok implementálásánál. Az SSJ egy Java könyvtár sztochasztikus szimulációk számára, melyet Pierre L'écuyer irányítása alatt fejlesztettek ki a Montreali Egyetemen. Lehetőséget nyújt egyenletes, és nem-egyenletes eloszlású véleltlen változók generálására, és diszkrét esemény szimulációra mind eseményekkel, mind folyamatokkal. Az SSJ hivatalos honlapja, ahonnan a keretrendszer, a forráskód, és a licenc is letölthető: http://www.iro.umontreal.ca/~simardr/ssj/indexe.html

3.1 A grafikus felhasználói felület kialakítása

Pár szó a JFC-ről, és a Swingről: A JFC (Java Foundation Classes) osztálykönyvtár tartalmazza a grafikus felhasználói felület (GUI) előállítására alkalmas osztályokat, melyekkel

gazdag grafikai megoldásokat és interaktivitást vihetünk alkalmazásainkba. A következő részeket tartalmazza:

Swing GUI komponensek	Ide tartozik minden a gomboktól kezdve a panelek felosztásán át a táblázatokig. Sok komponens sorba rendezhető, és nyomtatható, csak hogy pár támogatott tulajdonságot is megnevezzünk.
Cserélhető Look-and-Feel (*vizuális megjelenés*) támogatása	A Swing alkalmazások vizuális megjelenése cserélhető, ami lehetővé teszi az egyéni vizuális megjelenést. Például egy program használhatja a Java, vagy a Windows vizuális megjelenését is. Ráadásul a Java platform támogatja a GTK+ vizuális megjelenést is, ami lehetővé teszi, hogy több száz különböző vizuális megjelenés álljon a Swing programok rendelkezésére.
Hozzáférhetőség API	Engedélyezi olyan kisegítő technológiák számára a felhasználói interfészről származó információk hozzáférését, mint a képernyő olvasó, vagy a Braille megjelenítők eszköz
Java 2D API	Lehetővé teszi a fejlesztők számára, hogy könnyedén építsenek alkalmazásaikba jó minőségű 2D-s grafikát, szöveget, vagy képeket. A Java 2D átfogó API-kat tartalmaz ahhoz, hogy jó minőségű kimenetet készíthessünk, vagy küldhessünk nyomtató eszközök számra.
Nemzetköziség	Lehetőséget nyújt a fejlesztők számára, hogy olyan alkalmazásokat építsenek, amelyek a felhasználókkal a saját anyanyelvükön képesek kommunikálni, sőt, el tudják fogadni a japán, kínai stb. karaktereket is.

[9][10]

Ezen komponensek konkrét alkalmazását a *NetBeans Swing GUI Bulider* tette lehetővé. A *Swing GUI Builder*-nek köszönhetően a komponenseket csak egy palettáról kellett egy tervező keretbe belehúzni, és a grafikus építő automatikusan beállított az elrendezésüket. Ezzel a módszerrel alakítottam ki a program grafikus kezelő felületét.

3.2 Szimulációs megközelítések

Az SSJ alapvetően kétfajta szimulációs megközelítést támogat:

1. Folyamat-orientált implementáció
2. Esemény-orientált implementáció

3.2.1 Folyamat-orientált implementáció

A folyamat-orientált szimulációt az SSJ `simprocs` csomagja szolgáltatja. Egy folyamat tekinthető egy aktív objektumnak, melynek viselkedését az `actions()` metódusa írja le. Minden folyamatnak ki kell terjesztenie a `SimProcess` osztályt, és implementálnia kell az előbb említett `actions()` metódust. Ezek a folyamatok időzíthetőek, akárcsak az események, amikről később lesz szó. Viszont az eseményekkel ellentétben, nem feltétlenül azonnal hajtódik végre az `action()` metódusban foglalt viselkedés. Egy adott szimulációs időben legfeljebb egy folyamat lehet aktív. Egy aktív folyamat létrehozhat és időzíthet új folyamatokat, leállíthat felfüggesztett folyamatokat, vagy felfüggesztheti magát. Egy folyamat felfüggesztve marad, még le nem jár egy adott fix késleltetés, vagy szabaddá nem válik számára egy erőforrás, illetve még egy bizonyos feltétel igaz nem lesz. Ha egy folyamat felfüggesztett állapotba kerül, vagy befejezi működését, akkor általában egy másik folyamat indul el, vagy folytatódik.

Ezek a folyamatok olyasmi egyedi objektumoknak tekinthetőek, mint a gépek, vagy robotok egy gyárban, vásárlók egy boltban, vagy járművek egy szállító rendszerben. A folyamat-orientált paradigma a természetes módja annak, hogy hogyan írható le egy komplex rendszer, és gyakran tömörebb programkódot eredményez, mint az esemény-orientált megközelítés. A gyakorlatban viszont az esemény-orientált implementáció az elterjedtebb, mivel úgy gyorsabb szimulációs programok készíthetőek, mivel elhagyhatóak a folyamatok szinkronizálásával járó terhek. A legtöbb komplex diszkrét esemény rendszer is egyszerűen csak eseményekkel

vannak modellezve. Az SSJ-ben a folyamatok, és az események szabadon használhatóak felváltva, vagy együtt. [19]

3.2.2 Esemény-orientált implementáció

Az esemény-orientált megközelítésben minden igény passzív objektumként van jelen. Egy szimuláció folyamán minden bekövetkezhető esemény az `Event` osztály leszármazottja, például egy igény érkezése, vagy kiszolgálása, illetve maga a szimuláció vége is. Ha létrejön egy esemény példány, akkor az bekerül az *eseménysorba* a neki megfelelő időzítéssel, és akkor fog végrehajtódni, ha a szimulációs óra eléri ezt az időt. Egy esemény végrehajtása azt jelenti, hogy lefut az `actions()` metódusa. A szimulációs órát, és az eseménysort (azt a listát, mely tartalmazza a bekövetkezendő eseményeket) a `simevents` csomag `Simulator` osztálya kezeli a háttérben. Ez az osztály a diszkrét esemény szimuláció végrehajtója. [20]

3.2.3 A Qsim választása

A *Qsim* elsősorban az *esemény-orientált megközelítést* alkalmazza. A fejlesztés első fázisában a szimulációs motor folyamat alapúra lett tervezve, de a probléma az volt, hogy az SSJ folyamatvezérelt szimuláció megvalósítása a Java *zöld szálakra* lett kifejlesztve, amik alapvetően szimulált szálak, de csak a JDK 1.3.1, és régebbi verziókban találhatóak meg. Sajnos a legújabb virtuális gépek már nem támogatják ezeket, helyettük natív szálak vannak. Ez viszont korlátozza a folyamatok számát, lelassítja a szimulációt, és bizonyos esetekben gátolja a program megfelelő futását is. A *Qsim* tartalmaz egy folyamatvezérelt szimulációs motort is, de használata nem ajánlott, csak demonstrációs célra készült, hogy bemutassa ezt a megközelítést is.

3.3 A szimuláció paraméterezése

Ahhoz, hogy egy sorbanállási rendszer szimulálható legyen, meg kell adni a rendszer típusát, és a szimuláció hosszát. A rendszert típusától függően minden szimuláció előtt felépül a virtuális sorbanállási rendszer, melynek legfőbb logikai komponensei a következőek:

- beérkezési időközök eloszlása
- véletlen változó generátor a beérkezési időközökhöz
- kiszolgálási idők eloszlás

- véletlen változó generálása a kiszolgálási időkhöz
- a kiszolgálóhoz tartozó lista (*kiszolgálók száma*)
- várólista (*kapacitás szabályozható vele*)
- kiszolgálási elv
- szimuláció hossza (*időtartama*)

3.4 Eloszlások

Az SSJ `probdist` csomagja foglalkozik a valószínűségi eloszlásokkal. Eszközöket biztosít arra, hogy néhány folytonos, és diszkrét valószínűségi eloszlás sűrűség függvényei, súlyfüggvényei, eloszlásfüggvényei, azok inverzei számolhatóak legyenek. Továbbá arra is lehetőséget nyújt néhány eloszlásnál, hogy azok paraméterei meghatározhatóak legyenek empirikus adatokból. Véletlen szám generálásra viszont nincs lehetőség, arról a `randvar` csomag gondoskodik. [16]

Emlékeztetés képek egy X folyamatos valószínűségi változó eloszlás függvénye, melynek sűrűség függvénye f:

$$F(x) = P[X \leq x] = \int_{-\infty}^{x} f(s)ds$$

Diszkrét esetben, az X diszkrét valószínűségi változó eloszlása, melynek súlyfüggvénye p, és a véges $x_0 < x_1 < x_2 < \cdots$ valós halmazon van értelmezve:

$$F(x) = P[X \leq x] = \sum_{x_{1\leq}\, x} p(x_i)$$

ahol $p(x_i) = P[X = x_i]$. Egy olyan diszkrét eloszlásnak, mely az egész számok felett van értelmezve a következő eloszlásfüggvénye van:

$$F(x) = P[X \leq x] = \sum_{s=-\infty}^{x} p(s)$$

ahol $p(s) = P[X = s]$.

Az *X* komplementer eloszlás függvényét, $\bar{F}$-t úgy definiáljuk, hogy:

$$\bar{F}(x) = P[X \geq x]$$

Az $\bar{F}$ ilyen módú definíciójából adódik, hogy $\bar{F}(x) = 1 - F(x)$ a folytonos eloszlásoknál, és $\bar{F} = 1 - F(x-1)$ a diszkrét eloszlásoknál az egész számok halmazán. Ez a definíció diszkrét esetben viszont *nem sztenderd*, ugyanis $\bar{F}(x) = P[X \geq x]$ szerepel $\bar{F}(x) = P[X > x] = 1 - F(x)$ helyett. Viszont így néhány számítás sokkal egyszerűbben elvégezhető.

Az *inverz eloszlás függvény* a következő képen van definiálva:

$$F^{-1}(u) = \inf\{x \in \mathbb{R}: F(x) \geq u\}$$

úgy, hogy $0 \leq u \leq 1$. Ez az F^{-1} függvényt többek közt arra használják, hogy fordított módon generálják a véletlen *X*-et, úgy hogy az *u*-nak egy *U(0,1)* véletlen változót adnak értékül.

A `probdist` csomag kétféle lehetőséget kínál a *p, f, F,* és F^{-1} kiszámolására: *statikus metódusok*, amik használatához nincs szükség példányosításra, és olyan metódusok, amik egy *eloszlás objektumhoz* vannak társítva. Az egyes sztenderd eloszlások saját osztályokban vannak implementálva. Akkor érdemes létrehozni egy példányt, ha az *F,* és F^{-1} függvényt többször is ki kell számolni ugyanannál az eloszlásnál. Néhány estben (például a Poisson-eloszlásánál) a példányosítás során előre kiszámításra kerülnek bizonyos táblák, hogy jelentősen felgyorsítsák például az *F,* és F^{-1} függvények kiszámolását. Ez a sebességnövelés viszont több memória használatba, és egy egyszeri számítási költségbe kerül.

Az valós számok felett értelmezett eloszlásokat implementáló osztályok a `DiscreteDistribution` (diszkrét eloszlások), vagy a `ContinuousDistribution` (folytonos eloszlások) absztrakt osztályok egyikét terjesztik ki (mindkettő implementálja a `Distribution` interfészt). Diszkrét esetben a nem negatív egész számok felett az eloszlást implementáló osztály a `DiscreteDistributionInt` osztályt terjeszti ki.

3.4.1 Exponenciális eloszlás

Az $1/\lambda, \lambda > 0$ várható értékű exponenciális eloszlást a `ContinuousDistribution`-t kiterjesztő `ExponentialDist` osztály implementálja.

Az exponenciális eloszlás sűrűségfüggvénye:

$$f(x) = \lambda e^{-\lambda x}, \qquad x \geq 0$$

Eloszlásfüggvénye:

$$F(x) = 1 - e^{-\lambda x}, \qquad x \geq 0$$

Inverz eloszlásfüggvénye:

$$F^{-1}(u) = -\frac{\ln(1-u)}{\lambda}, \qquad 0 < u < 1$$

3.4.2 Erlang eloszlás

Az SSJ-ben az Erlang eloszlást implementáló osztály (`ErlangDist`) a `GammaDist` osztályból van származtatva, ugyanis az Erlang eloszlás a gamma eloszlás egy speciális esete, olyan módon, hogy a *k* paramétere egész szám.

Paraméterei:

$$k > 0, \lambda > 0 \qquad k \in \mathbb{Z}$$

Sűrűségfüggvénye:

$$f(x) = \frac{\lambda^k x^{k-1} e^{-\lambda} x}{(k-1)!}$$

Kumulatív eloszlás függvénye:

$$\frac{\gamma(k, \lambda x)}{(k-1)!} = 1 - \sum_{n=0}^{k-1} e^{-\lambda x} (\lambda x)^n / n!$$

3.4.3 Hiperexponenciális, és hipo-exponenciális eloszlások

Az SSJ ezeket az eloszlásokat nem tartalmazza. A *Qsim* számára készítettem két pszeudoosztályt (`PseudoHypoExpDist`, *és* `PseudoHyperExpDist`), melyek mindketten a `ContinuousDistribution`-ből lettek származtatva. Azért pszeudoosztályok, mert nem implementálják sem a sűrűségfüggvényeket, sem az eloszlásfüggvényeket, mivel ezeknek nem létezik zárt alakjuk. A gyakorlati hasznuk majd a

véletlen változó generálásnál lesz. A két osztály az implementáció tekintetében sem különbözik egymástól: mindketten egy-egy exponenciális eloszlásokat tartalmazó listával rendelkeznek. Azért lett mégis két különböző osztály létrehozva, hogy amikor a szimulációs motor megkap egy folytonos eloszlást, akkor el tudja dönteni, hogy melyik konkrét eloszlásról is van szó, mert annak tükrében konstruálja meg a véletlen változó generátort. Továbbá az eredményeket megjelenítő összefoglaló táblában így tudja a program felismerni, hogy milyen a beérkezési időközök, illetve kiszolgálási idő eloszlása, amikor kiírja a sorbanállási rendszer típusát.

3.5 Véletlen szám generátorok

Minden szimulációs programban központi szerepet játszanak a véletlen szám generátorok, ugyanis ezek biztosítják egy szimuláció eredményének statisztikai szempontból vett hitelességét. A véletlen szám generátorok segítségével állnak elő a futás során a beérkezési időközök, és a kiszolgálási idők. Így ezek nyilván *nem-egyenletes* véletlen változó generátorokat követelnek meg. Ezek előállítását az SSJ a `randvar`, és `rng` csomagjai teszik lehetővé.

Az `rng` csomag alapvető eszközöket biztosít egyenletes eloszlású valószínűségi változók generálására, többek közt a `RandomStream` interfészt, és annak néhány implementációját. Ez az interfész meghatározza, hogy minden véletlen szám folyamnak további alfolyamokra kell bontódnia, és a metódusoknak tudniuk kell az alfolyamok között váltaniuk.

Minden implementáció egy sajátos gerinc véletlen szám generátort (RNG) használ, melynek periódushossza nagyon hosszú egymást át nem fedő szegmensekre osztódik fel. Ezek a szegmensek biztosítják a véletlen folyamokat, és alfolyamokat. Egy folyam képes egyenletes eloszlású valószínűségi változókat generálni a (0,1) intervallumon, továbbá olyan egyenletes eloszlású valószínűségi változókat, melyek értékeit az {i, …, j} egész számsorozatból veszik fel.

A rendelkezésre álló RNG-k különböző sebességűek, és különböző periódus hosszal bírnak. Az *MRG32k3*a az egyik legalaposabban tesztelt, bár nem a leggyorsabb. Az *LFSR113, GenF2w32, MT19937,* és a *WELL* olyan bitsorozatokat generálnak, amikre igaz a lineáris

ismétlődés, így ezek megbuknak az olyan statisztikai teszteken, melyek a bitsorozatok lineáris komplexitását vizsgálják. Bár ez csak nagyon kevés speciális alkalmazásnál jelent problémát.

A következő táblázat minden RNG-re megadja a jellemzőket: (A mérés egy 2400 MHz-es 64bit-es AMD Athlon 64 4000+ processzoron történt Linux alatt, JDK 1.5.0-tel)

- a periódus hosszat (*periódus*)
- azt az időt másodpercekben, amire a CPU-nak szüksége van, hogy 10^9 U(0,1) véletlen számot generáljon (*generálási idő*)
- azt az időt másodpercekben, amire a CPU-nak szüksége van, hogy 10^6-szor előre ugorjon a következő alfolyamra (*ugrási idő*)

RNG	periódus	generálási idő	ugrási idő
LFSR113	2^{113}	31	0,08
WELL607	2^{607}	33	329
WELL512	2^{512}	33	234
WELL1024	2^{1024}	34	917
LFSR258	2^{258}	35	0,18
MT19937	2^{19937}	36	46
GenF2w32	2^{800}	43	556
MRG31k3p	2^{185}	51	0,89
F2NL607	2^{637}	65	329
MRG32k3a	2^{191}	70	1,1
RandRijndael	2^{130}	127	0,6

.

3.5.1 Véletlen folyamok

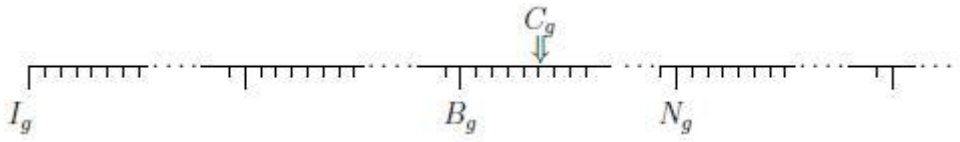
Minden alaptípusú véletlen szám generátor periódusa *Z* hosszúságú szomszédos folyamra (vagy szegmensre) bomlik Minden ilyen folyam pedig *V* alfolyamra osztódik, melyek hossza *W*, így *Z=VW*. A *V* és *W* értékek a konkrét véletlen szám generátortól függenek, de általában nagyobbak, mint 2^{50}. Így a *Z* távolság a véletlen szám által biztosított két szomszédos folyam kezdőpontja között általában meghaladja a 2^{100}-t. A véletlen szám generátor kezdeti *seed*-je az első folyam kezdőpontja. Minden RNG-nek van alapértelmezett *seed*-je, de az

megváltoztatható. Minden alkalommal, amikor egy új `RandomStream` objektum létrejön, akkor a folyam kezdeti pontja (kezdeti *seed*-je) automatikusan úgy számolódik ki, hogy az *Z* lépéssel előrébb legyen az előzőleg létrehozott ugyan ilyen típusú folyam kezdőpontjához képest.

Minden folyamra a következők érvényesek:

- léphetünk egyet előre, és generáljuk az új értéket
- előre ugorhatunk a következő ezen folyamhoz tartozó alfolyam elejére
- visszaugorhatunk a jelenlegi alfolyam elejére
- visszaugorhatunk a folyam elejére
- tetszőleges számú lépést ugorhatunk előre, vagy hátra

Egy példán keresztül ismertetem a fentieket. Jelölje C_g a g folyam jelenlegi állapotát. I_g a kezdeti állapotot, B_g a jelenlegi alfolyam kezdetét, N_g pedig a következő alfolyam kezdetét. Az alábbi ábra egy olyan folyamot mutat, aminek jelenlegi állapota a 3. alfolyam 6. pozíciójában van, azaz $2W + 5$ lépésre a kezdőállapottól, és 5 lépésre a B_g pozíciótól. Egy folyam állapotának leírása a folyam típusától függ. Például egy *MRG32k3a* folyamának az állapota egy 6 elemű vektorral írható le, ami 32 bites egészeket tartalmaz, melyek belső ábrázolása viszont lebegőpontos. [18]



Ábra 1.

3.5.2 MRG32k3a

Az *MRG32k3a* fő generátorának a L'Ecuyer által javasolt 64 bites lebegőpontos aritmetikával implementált CMRG-t (*Combined Multiple Recursive Generator*) használja. Ennek a gerinc generátornak a periódus hossza $p \approx 2^{191}$. A *V, W* és *Z* értéki pedig rendre 2^{51}, 2^{76}, és 2^{127}. A véletlen szám generátor *seed*-jét, és bármely pontban a folyam állapotát egy 6 dimenziós vektor írja le, melynek elemei 32 bites egész számok, `double`-ben tárolva. Az alapértelmezett *seed* a következő: (12345, 12345, 12345, 12345, 12345, 12345).

A *Qsim* programozása során ezt a véletlen szám generátort választottam minden esetben, ahol `RandomStream` objektumra volt szükség, vagy az [0,1)-en szerettem volna egy véletlen számot generálni. Minden példányosítás után módosítottam az alapértelmezett *seed*-et, úgy, hogy a Java `Math` osztályának `random()` metódusával generáltam egy véletlen számot a (0,1)-en, azt megszoroztam 100000-el, majd szintén a `Math` osztály `round()` metódusával pedig egészre kerekítettem. Így elértem azt, hogy minden futtatás alkalmával más, és más legyen a véletlen szám generátor által generált számok sorozata.

3.5.3 Erlang eloszlású véletlen változó generálása

Az SSJ `randvar` csomagja biztosítja a *nem-egyenletes* eloszlású valószínűségi változók generálását. A *Qsim* lehetőséget nyúlt Erlang eloszlású beérkezési időközök, illetve kiszolgálási idők szimulálására. Ehhez egy olyan véletlen szám generátorra van szükség, ami ilyen eloszlású véletlen változókat generál.

A `RandomVariateGen`, és a `RandomVariateGenInt` két olyan általános osztály, melyek lehetővé teszik véletlen változó generátorok létrehozását egy véletlen szám folyamból, és egy tetszőleges eloszlásból. Ahhoz, hogy egy valós számok feletti valószínűségi változót generáljunk egy tetszőleges eloszlás inverziója által, azt a következő képen tehetjük az SSJ alatt: Létre kell hozni egy új `RandomVariateGen` objektumot 2 paraméterrel, a `Distribution` interfészt implementáló megfelelő osztály példányával, és egy `RandomStream` objektummal, ami a véletlen folyamot fogja biztosítani. Ez után, ha véletlen számokat akarunk kapni, akkor a `RandomVariateGen nextDouble()` metódusát kell meghívni, annyiszor ahányszor szükségünk van rá. [17]

Erlang eloszlású valószínűségi változót úgy kaphatunk, ha létrehozunk egy `RandomVariateGen` példányt, és a konstruktorában az Erlang eloszlást implementáló `ErlangDist` egy példányát adjuk át. Lehetne úgy is Erlang eloszlású véletlen változókat generálni, hogy közvetlenül az `ErlangGen` osztályt példányosítjuk, de a *Qsim* felépítésből adódóan, az előbbi megoldás kézenfekvőbb, hiszen nem kell vizsgálni az paraméterül kapott eloszlás típusát (jelen esetben ez természetesen az Erlang).

3.5.4 Exponenciális eloszlású véletlen változó generálása

Az előz bekezdésben ismertetettek alapján új exponenciális eloszlású véletlen változó generátort is hasonló képen kell létrehozni, mint Erlang eloszlásút. Itt látszik, hogy új szimuláció során elég csak az eloszlást átadnia a szimulátor motorjának, hiszen annak nem kell tudni arról, hogy milyen eloszlásról is van szó, mivel minden osztályból, mely implementálja a `Distribution` interfészt, előállítható véletlen változó generátor.

3.5.5 Hipo-exponenciális eloszlású véletlen változó generálása

Hipo-exponenciális eloszlású véletlen változó nem generálható, olyan módon, mint az előbb ismertetett két véletlen változó generátor. Ez abból adódik, hogy nem konstruálható olyan osztály, amely a hipo-exponenciális eloszlást implementálná. Ahhoz, hogy egy osztály eloszlásosztály legyen, ahhoz implementálnia kell a `Distribution` interfészt. Viszont ez az interfész olyan metódusokat követel meg, mint például az $F(x)$, vagy az $F^{-1}(u)$ meghatározása, amik ennél az eloszlásnál elég bonyolultak, ráadásul a programnak nincs is mindegyikre szüksége.

Így hipo-exponenciális eloszlású véletlen változót másképpen kell előállítanunk: úgy hogy közvetlenül a `RandomVariateGen` osztályt terjesztjük ki. Ez viszont azzal a kellemetlenséggel jár, hogy amikor a szimulációs motor megkapja a szimuláció paramétereit, akkor figyelnie kell, hogy a kapott eloszlás a `PseudoHypoExpDist` osztály példány-e, mert akkor más véletlen változó generátor osztályt kell létrehozni, nem az általános `RandomVariateGen`-t. Ezt a véletlen változó generátor osztály pedig a `HypoExpRandomVariateGen` implementálja.

A véletlen változó generálásának elve, ha a hipo-exponenciális eloszlás i különböző intenzitású exponenciális paraméterrel rendelkezik:

$$\eta \in U(0,1)$$

$$\xi_i = -\frac{\ln(1-\eta_i)}{\lambda} \in Exp(\lambda_i)$$

$$\sum \xi_i \ a\ v\acute{e}letlen\ v\acute{a}ltoz\acute{o}$$

3.5.6 Hiperexponenciális eloszlású véletlen változó generálása

A hiperexponenciáls eloszlás is hasonló helyzetben van, mint a hipo-exponenciális. Ezt is csak egy pszeudoosztály implementálja, pontosan olyan okok miatt, mint amik a hipo-exponenciális eloszlásnál voltak. Hiperexponenciális eloszlású véletlen változót a `HyperExpRandomVariateGen` osztály generál a következő elvet követve:

A hiperexponenciális eloszlás sűrűségfüggvénye, mely *n* különböző intenzitású exponenciális paraméterrel rendelkezik:

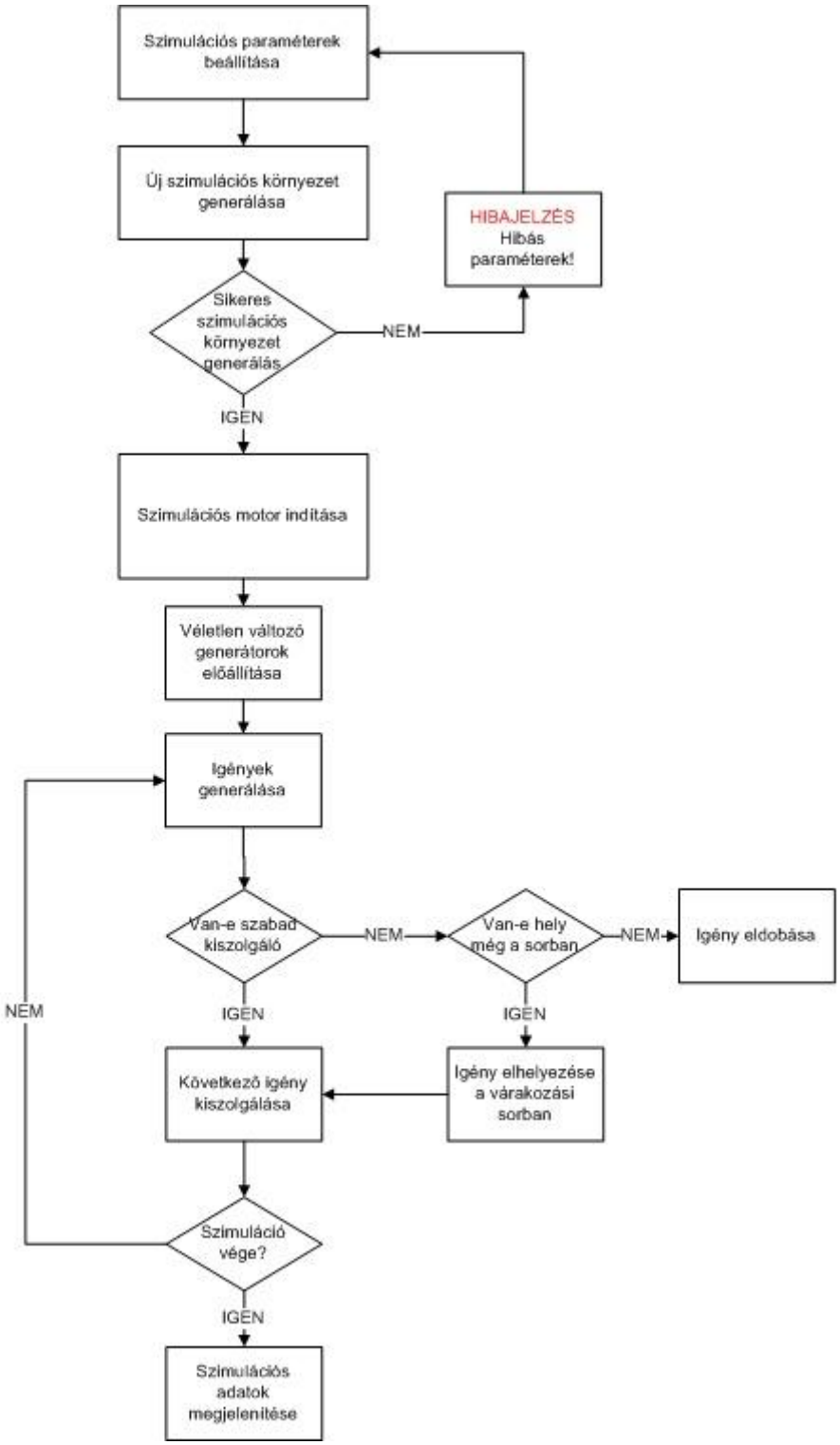
$$f(x) = \sum_{i=1}^{n} p_i(1 - e^{-\lambda_i x})$$

$$\sum_{i=1}^{n} p_i = 1$$

A p_i azt jelzi, hogy a hiperexponenciális valószínűségi változó milyen valószínűséggel veszi fel az *i*. exponenciális eloszlása formáját.

A véletlen változó generálása ezek alapján pedig a következőképpen lett implementálva: A `HiperExpRandomVariateGen` rendelkezik egy listával, amiben exponenciális eloszlású véletlen változó generátorok vannak. Új véletlen szám generálásakor generál az (1,n)-en egy egyenletes eloszlású véletlen változót, majd a kapott véletlen számot egészre kerekíti. A listából pedig ennek a sorszámú exponenciális eloszlású véletlen változó generátorának adja vissza az új véletlen számát.

3.6 A szimuláció menete



Ábra 2.

A fenti ábrán látható a *Qsim* szimulációjának logikai menete. Ebben a fejezetben a szimulációs folyamat főbb lépéseit fogom taglalni, melyek:

- a szimulációs környezet inicializálása
- a szimulációs motor működése
- a szimulációs folyamat eredménye

3.6.1 A szimulációs környezet inicializálása

Amikor a *Qsim* programmal egy új szimulációt indítunk, akkor először be kell olvasnunk a grafikus felhasználói felületről a kívánt sorbanállási rendszer típusát, és az egyéb szimulációs paramétereket. A *Szimuláció indítása* gomb lenyomása után a program értelmezi a beállított paramétereket. Megpróbálja létrehozni a beérkezési időközöket, és a kiszolgálási időket reprezentáló konkrét eloszlás példányokat. Amennyiben ez a folyamat kivételt dob, akkor a *Qsim* ezt egy hibaüzenetben jelzi a felhasználó felé. Ez általában akkor fordul elő, ha tört számot adunk meg olyan helyen, ahol egészet várna a program. Az ilyen okokból adódó problémák csökkentésére a rendszer folyamatosan figyeli a beviteli mezőket, és csak megfelelő értékeket engedélyez bevinni. Ez csak egyike a sok érvényességet vizsgáló funkciónak. A többiről *A Qsim program használata* című fejezetben lehet olvasni.

Ha minden paraméterből érvényes változót tudott előállítani a program, akkor azokkal meghívja az eredmények megjelenítéséért felelős programrészt: példányosít egy új `SimResultInTable` osztályt. Ennek az osztálynak a fő feladata, hogy a kapott paraméterekkel lefuttasson egy szimulációt, tárolja annak kimenetelét, és megjelenítse annak eredményét. Ezt úgy valósítja meg, hogy létrehoz egy új `QueueMotor` példányt. Ez az osztály egy absztrakt osztály. Két osztály terjeszti ki: a `QueueProc`, mely a folyamat-orientált megközelítés szerint végzi a szimulációt, és a `QueueEv` osztály, mely az esemény-orientált paradigmát követi. Mivel a *Qsim* technikai okokból az utóbbit támogatja, így a programban a `QueueMotor` típusú változó értéke egy `QueueEv` példány lesz. Ez a példány végzi a tényleges szimulációt.

3.6.2 A szimulációs motor működése

A `SimResultInTable` létrehozza a `QueueEv` egy példányát a megfelelő paraméterekkel, melyek*: beérkezési időközök eloszlása, kiszolgálási idők eloszlása, kiszolgálók száma,*

kapacitás, kiszolgálási elv, és *a szimuláció hossza*. A `QueueEv` ezekből az adatokból pedig kialakítja a szimulációs környezetet:

- eltárolja a beérkezési időközök eloszlását, és a kiszolgálási idők eloszlását
- az eloszlásoknak megfelelően létrehozza a beérkezési időközökhöz tartozó véletlen változó generátort (`arrGen`), és a kiszolgálási időkhöz tartozó véletlen változó generátort (`servGen`)
- létrehozza a várakozási sort, és azt a listát, amely a kiszolgálókat reprezentálja
- eltárolja a paraméterül kapott többi adatot is
- létrehozza az üres statisztikai mintagyűjtőket a *kihasználtság, sorhossz, várakozási idők, kiszolgálási idők, válaszidők,* és az *igényszámláló* számára

Az SSJ esemény-orientált implementációját biztosító eszközök a `simevents` osztályban találhatóak. Minden olyan osztály, amely implementálja az `Event` osztályt, egy a szimuláció során bekövetkezhető eseménynek tekinthető. Jelen esetben ezek a következőek:

- igény érkezése (`Arrival` osztály)
- igény kiszolgálása (`Departure` osztály)
- új statisztikai megfigyelés (`NewObservation` osztály)
- szimuláció vége (`EndOfSim` osztály)

Minden esemény, amikor létrejön, bekerül az *eseménysorba* a tervezett bekövetkezésének idejével. Ha a szimulációs óra eléri ezt az értéket, akkor az esemény végrehajtódik. Egy esemény végrehajtása azt jelenti, hogy meghívásra kelül az `actions()` metódusa, így minden eseménynek implementálnia kell ezt a metódust. A szimulációs órát, és az eseménysort (*az a lista, mely tartalmazza a jövőbeli eseményeket*) a háttérben a `Simulator` osztály kezeli. Ez tekinthető a diszkrét esemény szimuláció végrehajtójának. Metódusai segítségével indítható, újraindítható, illetve megállítható a szimuláció. Továbbá leolvasható a szimulációs óra értéke. A *Qsim* (és a legtöbb program is) általában csak egy szimulációs órát használ. Így az egyszerűség kedvérét a `Simulator` osztály biztosít egy alapértelmezett szimulátort, amit az olyan események használhatnak, melyek létrehozása során nem adunk meg explicit hivatkozást egy konkrét `Simulator` példányra.

A szimuláció akkor kezdi meg futását, ha a `QueueEv simulateOneRun()` metódusa meghívásra kerül. Ezt a metódust a `SimResultInTable` hívja meg rögtön az után, hogy példányosította a szimulációs motort – azaz a `QueueEv` osztályt. A metódus a megadott rendszert a paraméterként megkapott szimulációs idő végéig szimulálja. Első lépésként inicializálja a szimulációs órát, és az eseménysort. Az eseménysorban elhelyezi az `EndOfSim` eseményt (szimulációt megállító esemény), egy `NewObservation` eseményt, és az első igény érkezését jelentő `Arrival` eseményt. Végezetül elindítja a szimuláció, úgy hogy az alapértelmezett szimulátort `start()` metódusát hívja. Ez a `start()` metódus pedig úgy kezdi el a szimuláció, hogy az eseménysorban lévő első esemény bekövetkezésének idejére állítja az órát, kiveszi ezt az eseményt a listából, és végrehajtja. Ezt addig folyatja, még ki nem ürül az eseménysor, vagy nem történik egy `stop()` hívás. Jelen esetben a kezdő eseménysorban a fentebb említett eseményekből 3 van, a következő ütemezésekkel:

- `EndOfSim`: a szimulációs idő hossza
- `NewObservation`: 1/10-ed időegység
- `Arrival`: az arrGen által generált véletlen szám

Az `Arrival` osztály `actions()` metódusa írja le, hogy mi is történik egy érkezés esetén. Az érkezések dominó elv alapján vannak ütemezve: minden érkezés esemény első dolga, hogy az `arrGen` által generált idővel ütemezi a következő érkezés eseményt. *(Így valósul meg az, hogy a beérkező időközök eloszlása olyan, mint amilyennel az `arrGen` véletlen változó generátor lett konstruálva, azaz amilyet mi adtunk meg a sorbanállási rendszer kiválasztásakor)*. Ezután létrehoz egy új igényt, melyben eltárolja a jelenlegi szimulációs időt. Ezt követően pedig növeli az igényszámlálót. Ha az összes kiszolgáló foglalt, azaz a kiszolgáló listában annyian vannak, mint a kiszolgálók száma, akkor az igény megpróbál beállni a várakozási sorba. Azonban, ha a rendszer kapacitása véges, és az igény már nem fér be a sorba, akkor egyszerűen elvész. Ha viszont beáll a sorba, akkor megkapja a majdani kiszolgálási idejét, melyet a `servGen` generál számára. Mivel ez nem folyamat-orientált megközelítése a szimulációnak, így már most is meg lehet határozni, hogy mennyi időt fog eltölteni a kiszolgálónál. Ha sorbaállás történik, akkor a sorhosszhoz rendelt statisztikai figyelő új megfigyeléssel bővül éppen az előtt, hogy az igény beállna a sorba. Amennyiben van szabad kiszolgáló, az új igény bekerül a kiszolgálási listába, és ütemez egy távozási eseményt, annyi idővel későbbre, amennyi a számára meghatározott kiszolgálási idő volt.

Ebben az esetben a várakozási időt, és a sorhosszat figyelő statisztikai mintavételezők 0 várakozási idővel, illetve sorhosszal bővítik a megfigyeléseiket. Továbbá a kiszolgálási időket mérő statisztikai mintavételező is frissül az igény kiszolgálási idejével.

Ha egy távozás esemény történik (amit a `Departure` osztály implementál), akkor egy igény távozik a kiszolgálási listából, és eltűnik. Ha a várakozási sor nem üres, akkor a kiszolgálási elvtől függően vagy az első, vagy az utolsó igény a várakozási sorból a kiszolgálási listába kerül át, és ütemezi a saját távozását. Ezen igény várakozási ideje bekerül a várakozási időt mérő statisztikai mintavételezőbe. A várakozási idő úgy kapható meg, hogy a szimulációs óra aktuális állásából kivonjuk az igény által tárol érkezési időt. A válaszidő pedig a várakozási idő, és kiszolgálási idő összege.

3.6.3 A szimulációs folyamat eredménye

A szimuláció végét az `EndOfSim` esemény bekövetkezte jelenti. Mivel a `simulateOneRun()` metódusnak nincs visszatérési értéke, így a szimulációs folyamat eredménye az, hogy a statisztikai mintavételezők feltöltődtek a szimuláció folyamán mért adatokkal. Ezek a figyelők a következő tulajdonságokról gyűjtöttek adatokat:

- kihasználtság
- sorhossz
- várakozási idő
- kiszolgálási idő
- válaszidő
- igényszámláló

Az igényszámláló kivételéve ezek a figyelők az SSJ `stat` csomagjának `TallyStore` osztállyal valósultak meg. Az igényszámláló csak egy egyszerű számláló. A `TallyStore` $X_1, X_2, X_3, \ldots$ megfigyeléseket tartalmaz. Ezekből az értékekből meg tudja mondani a minimumot, a maximumot, az átlagot, a szórást, és számolni tud konfidencia intervallumot is. A konfidencia intervallum számítása az alábbi statisztikán alapul:

$$T = \frac{\bar{X}_n - \mu}{S_{n,x}/\sqrt{n}}$$

ahol *n* a megfigyelések száma, $\bar{X}_n$ a megfigyelések átlaga, $S_{n,x}$ pedig az empirikus szórás. Ha feltesszük, hogy az *X* megfigyelések függetlenek, és azonos eloszlásúak, valamint normális eloszlásúak, akkor a *T* Student eloszlású *n-1* szabadsági fokkal. Ez által a módszer által adott konfidencia intervallum csak akkor hiteles, ha az előbbi feltevés teljesül, vagy ha *n* elég nagy ahhoz, hogy $\bar{X}_n$ körülbelül normális eloszlású legyen. [21]

A konfidencia intervallum meghatározására viszont a *Qsim* mégsem ezt a módszert választotta, hanem a *batch-means method*-ot, melyet a következő fejezetben részletezek.

3.7 Batch-means method

A *batch-means method* célja, hogy konfidencia intervallumbecslést adjon az $\mathcal{X} = \{X_i : i \geq 1\}$ diszkrét idejű stacionárius folyamat μ átlagára. [12] Tegyük fel, hogy egy diszkrét esemény szimulációs ideje alatt a folyamatról *m* megfigyelés készül, $X_1, \ldots, X_m$. Definiáljuk a sorozathoz tartozó mintaátlagot a következő képen:

$$\bar{X}_m = m^{-1} \sum_{i=1}^{m} X_i$$

Az *m* db mintát felosztjuk egymást nem fedő *n* db *b* hosszúságú alintervallumra, úgy hogy $m = b \cdot n$. Jelölje Y_i az *i*-edik intervallumhoz tartóz átlagot, azaz

$$Y_i = b^{-1} \sum_{j=1}^{b} X_{(i-1)b+j}, \qquad i = 1, \ldots, n$$

A *batch-means method*-ban az *n* a kötegek számát jelöli, *b* a köteg méretet, és az Y_i-k a kötegátlagokat. Definiáljuk az $\{Y_i\}$ empirikus szórásnégyzetét a következő módon:

$$V_{n,b} = n^{-1} \sum_{i=1}^{n} (Y_i - \bar{X}_m)^2$$

majd legyen

$$t_{n,b} = \frac{\bar{X}_m - \mu}{(V_{n,b}/n)^{1/2}}$$

Fix *n*-re gyenge szabályszerűségi feltételek mellett a $t_{n,b}$ valószínűségi változó eloszlása az $n - 1$ szabadságfokú Student-*t* eloszlás felé konvergál, ahogy $m \to \infty$. A Student-*t* eloszlás

függvénye pedig a sztenderd normális eloszlás függvénye felé konvergál, ahogy a szabadságfoka nő. Így használhatjuk a Student-*t* eloszlás függvényének, és a sztenderd normális eloszlás függvényének is a kvantilis pontjait a μ konfidencia intervalluma becslésére. Például ha z_α a sztenderd normális eloszlás függvényének α-kvantilis pontja, akkor a μ $100(1-2\alpha)\%$-os batch means (normális) konfidencia intervalluma: [11]

$$[\bar{X}_m + z_\alpha \left(\frac{V_{n,b}}{n}\right)^{\frac{1}{2}}, \bar{X}_m - z_\alpha \left(\frac{V_{n,b}}{n}\right)^{\frac{1}{2}}]$$

3.7.1 Batch-means method a *Qsim*-ben

A *Qsim* lehetőséget nyújt a *3.6.3*-as pontban ismertetett statisztikai mintavételezők megfigyeléseinek vizuális megjelenítésére. Ennek alapja az SSJ `charts` csomagja, melynek segítségével grafikonok konstruálhatóak. Ezek a grafikonok a megfigyeléseket többnyire a beérkező igények függvényében ábrázolják, kivéve a kihasználtságot, mert az az idő függvényében ábrázolja a foglalt kiszolgálók számát.

Ezekhez a megfigyelésekhez a program a *batch-means method* szerint számol konfidencia intervallumot. A felhasználó megadhatja, hogy hány százalékos konfidencia intervallumot szeretne megjeleníteni.

Kiemelném, hogy ennek a módszernek az alkalmazása teszi egyedivé a *Qsim*-et, ugyanis ezt a módszert nem szokták hasonló programokban implementálni.

4. A *Qsim* program használata

Az előző fejezetekben a *Qsim* elméleti, és elvi működéséről volt szó, amiből nem derült ki pontosan, hogy mire, és hogyan kell a programot használni. Ebben a fejezetben ismertem a program lehetőségeit, és azok helyes használatát.

A *Qsim* megtalálható a szakdolgozat CD mellékletén. Futtatásához legalább JVM (*Java Virtual Machine)* 1.5-ös verzió szükséges, de az egyszerűbb futtatás érdekében a JVM 1.6-os ajánlott. A program szerkezetét tekintve az SSJ könyvtárait tartalmazó `lib` könyvtárból, és a `Qsim.jar` fájlból áll, ami nem más, mint egy Java osztályokat, és meta adatot tartalmazó *zip* fájl. Ha nincs semmilyen program hozzárendelve (általában tömörítő programok szoktak) ehhez a kiterjesztéshez, akkor az 1.6-os JVM *executable jar file*-ként jelöli meg, azaz végrehajtható *jar* fájlként.

4.1 A *Qsim* indítása Windowsos környezetben

A *Qsim* indításához a program mappájában található `Qsim.jar` fájlt kell indítani. Amennyiben a Windows ezt a fájlt *executable jar file*-ként látja, akkor a `Qsim.jar` fájl futtatásával elindul a program. Ha a Windows alapértelmezettként nem a *Java(TM) Platform SE binary*-t társítja a *jar* fájlokhoz, de a fájlra jobb egérgombbal kattintva előugró menüben a *Megnyitás mással* menüpont alatt szerepel a *Java(TM) Platform SE binary* elem, akkor azt választva a *Qsim* indítható.

Amennyiben az előbbi esetek egyike sem áll fent, akkor a programot parancssorból kell indítani:

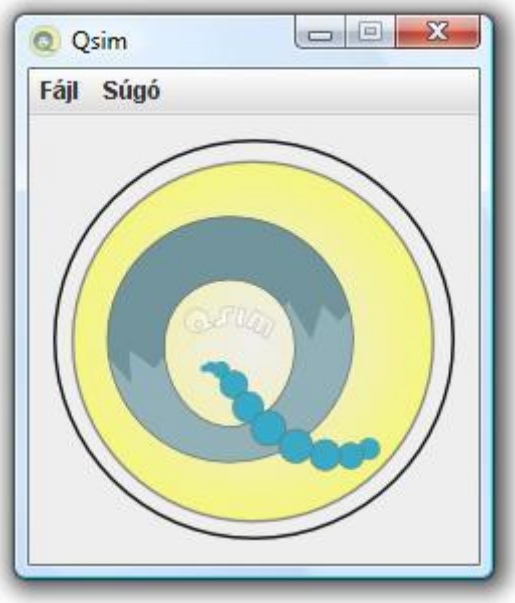
1. Indítsuk el a Windows parancssort (*például: Start menü→Futtatás→cmd*)
2. Lépjünk a *Qsim* könyvtárába
3. a `java –jar Qsim.jar` paranccsal a program indítható

4.2 A *Qsim* indítása Linuxos környezetben

A *Qsim* Linuxos környezetben való futtatáshoz indítsunk egy konzolalkalmazást. Lépjünk a programot tartalmazó könyvtárba, majd a `java -jar Qsim.jar` paranccsal a program elindítható.

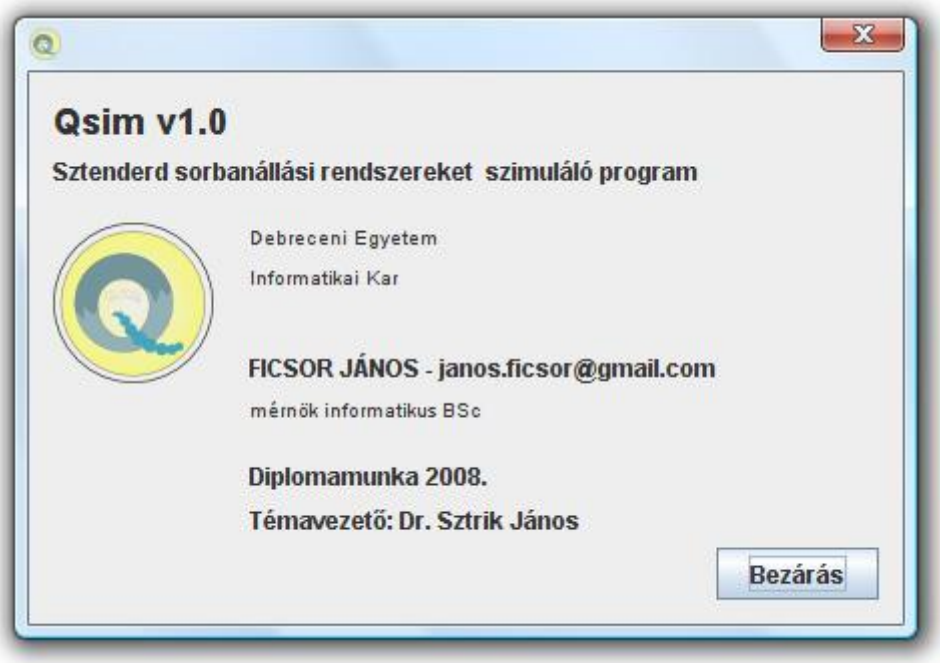
4.3 Új szimuláció indítása

A *Qsim* indítását követően a program főablaka hasonlóképpen néz ki: (*A Java Swing Look-and-feel-jétől függően – ez egy Winodws Vista Look-and-feel*)



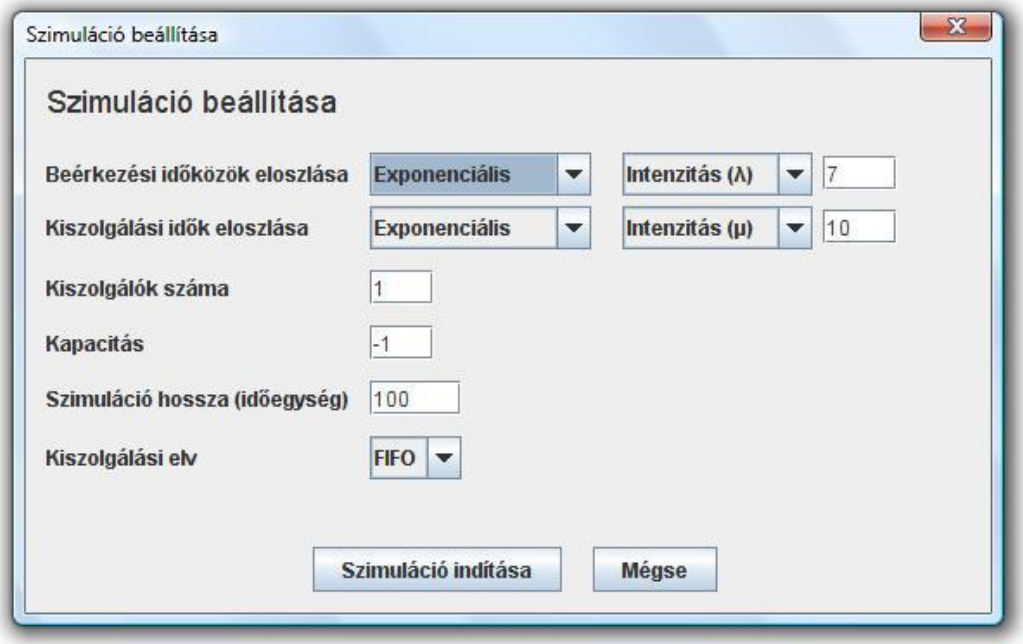
Ábra 3.

A főablak két főmenü ponttal rendelkezik. A *Súgó→Névjegy* pontja előhozza a program névjegyét, mely a program funkciójáról, és készítőjéről ad információt:



Ábra 4.

Új szimuláció a *Fájl* menü *Új szimuláció* pontjával indítható. Az indítás után a Java inicializálja a grafikus felhasználó felületet, ami után megjelenik a *Szimuláció beállítása* dialógusablak:



Ábra 5.

Ezen dialógus segítségével állíthatjuk be a szimulálni kívánt sorbanállási rendszer típusát.

4.3.1 Eloszlások beállítása

A *Qsim*-ben 4 féle eloszlás közül választhatunk a beérkezi időközök, illetve kiszolgálási idők eloszlásának megadásához, mégpedig:

- *Exponenciális eloszlás* λ paraméterrel (az eloszlás paraméterének jelölése különbözik attól függően, hogy a beérkezési időközök eloszlásáról, vagy a kiszolgálási időkről eloszlásáról van-e szó – ebben az esetben, ha kiszolgálási idők exponenciális eloszlásúak, akkor a paraméter a μ jelölést kapja)
- *Erlang eloszlás* k, és λ paraméterrel
- *Hipo-exponenciális eloszlás* λ_i paraméterekkel($i \geq 2$)
- *Hiperexponenciális eloszlás* λ_i paraméterekkel($i \geq 2$)

Az eloszlások beállításának első lépese, hogy kiválasztjuk a listából az eloszlásokat. Amint ez megtörtént, az eloszlás melletti paraméter listában a kiválasztott eloszlásra jellemző paraméterek jelennek meg. Exponenciális, és Erlang eloszlás esetében a paraméterek listája fix, viszont a hipo-exponenciális, és a hiperexponenciális eloszlásnál tetszőleges számú paramétert adhatunk meg. Ezt úgy oldja meg a program, hogy a paramétereket tartalmazó lista utolsó elem egy *Új λ...* értékű elem, melyre kattintva a program egy új paramétert ad a listához, és létrehozása után azt választja ki a felhasználó számára.

Az eloszlások paraméterezésére valós számok használhatók. Amennyiben a felhasználó mást próbál bevinni a paramétermezőkbe (például szöveget), akkor a program kitörli azt, és egy alapértelmezett értékét állít vissza a paramétermezőbe. Erre azért van szükség, hogy csökkenjen a hibás formátumú paraméterezésből adód sikertelen szimulációk száma. Bár még így is előfordulhatnak, hiszen egy Erlang eloszlást tudunk érvénytelenül paraméterezni, ha a *k*-nak tört számot adunk.

4.3.2 Kiszolgálók számának beállítása

Ezzel a paraméterrel a sorbanállási rendszer kiszolgálóinak száma adható meg. A paraméter értéke csak egész szám lehet. Tegyük fel, hogy a szimulálni kívánt sorbanállási rendszer kapacitása nem végtelen. Ebben az esetben rendelkezik egy beállított paraméterrel. Amennyiben szeretnénk megváltoztatni a kiszolgálók számát úgy, hogy az kisebb legyen,

mint a beállított kapacitás, akkor a program automatikusan átállítja a kapacitás paraméterét annyira, mint amennyi a kiszolgálók száma, hiszen egy rendszer kapacitása csak nagyobb vagy egyenlő lehet a kiszolgálók számánál.

4.3.3 Kapacitás beállítása

Ha olyan rendszert szeretnénk szimulálni, mely nem végtelen kapacitású, akkor annak kapacitása itt állítható be. A kapacitás csak olyan pozitív egész szám lehet, ami nagyobb, vagy egyenlő a kiszolgálók számával. Más érték esetén a rendszer visszaállítja az alapértelmezett kapacitás értékét, -1-et, melynek jelentése az, hogy a rendszer kapacitása végtelen. Új értéket úgy célszerű megadni, hogy kijelöljük a régit, és annak helyére írjuk az új értéket, ugyanis, hogy ha a -1 értéket úgy akarjuk módosítani, hogy először kitörüljük az 1-est, akkor a rendszer a mínuszjelet érvénytelen értéknek tekinti, és visszaállítja a -1-et.

4.3.4 Szimuláció hosszának beállítása

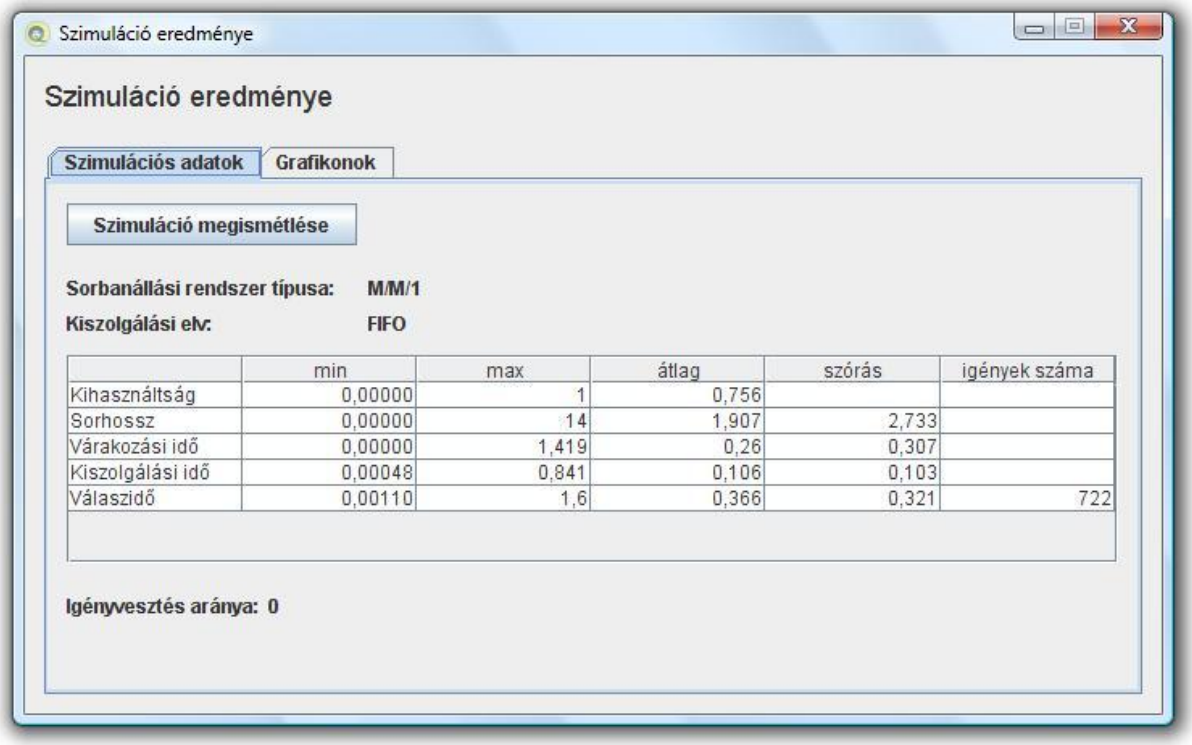
Ez a paraméter határozza meg, hogy meddig tartson a szimuláció, más szóval ez határozza meg a szimuláció hosszát. Értéke csak pozitív egész szám lehet.

4.3.5 Kiszolgálási elv

A kiszolgálási elvet a kiszolgálási elvek listájából lehet kiválasztani. A szimulálni kívánt sorbanállási rendszer a beállított kiszolgálási elvet fogja követni. A *Qsim* a FIFO-t, és a LIFO-t támogatja.

4.4 A szimulációs folyamat értékelése

Miután megnyomtuk a *Szimuláció indítása* gombot a *3.6* pontban ismertettek alapján lefut a szimuláció. A sikeres szimuláció után bezárul a szimuláció beállítás dialógus, és helyette megjelenik a szimuláció eredményét közlő ablak:

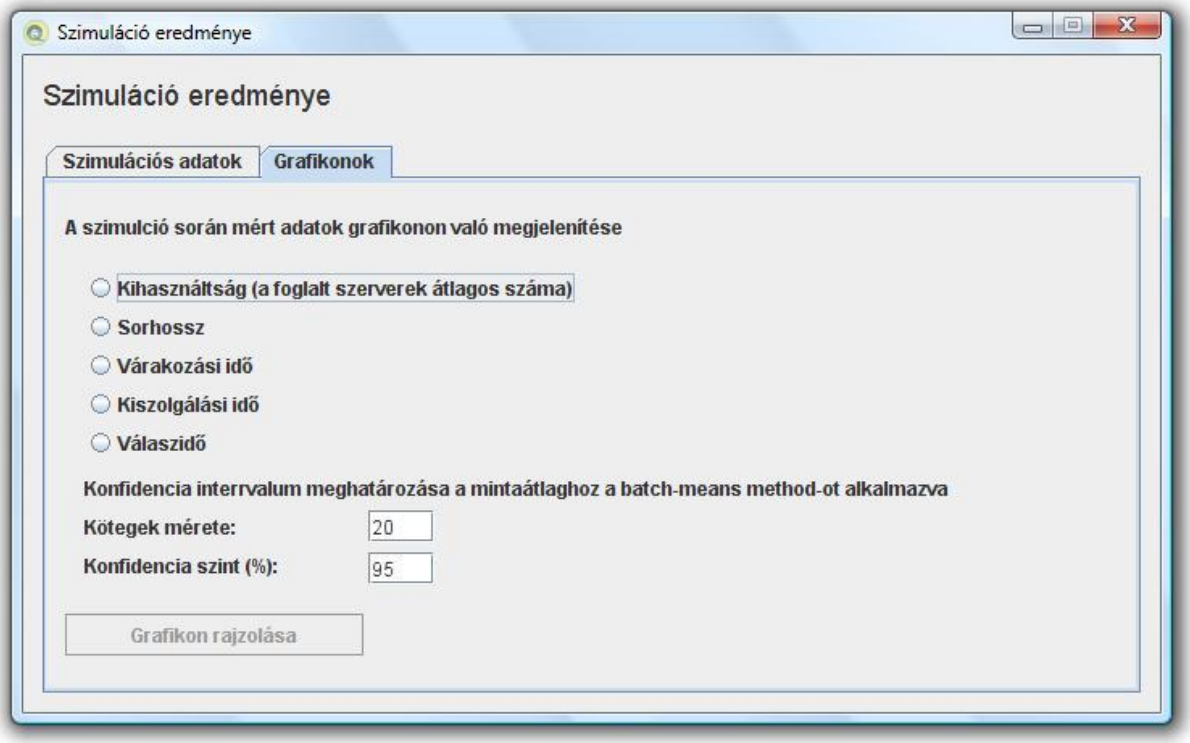


Ábra 6.

Az eredményt megjelenítő ablak két füllel rendelkezik. A *Szimulációs adatok* fül a statisztikai mintavételezők által számolt statisztikát jeleníti meg a 6-os ábrán látható táblázat segítségével. Az ablakban továbbá megjelenik a sorbanállási rendszer típusa is a Kendall-féle jelölést használva. Legalul látható az igényvesztés arány.

A *Szimuláció megismétlése* gomb hatására egy új szimuláció fut le az előzővel megegyező paraméterekkel, ezzel megismételve a szimulációs folyamatot. Mivel a véletlen szám generátorok minden inicializálásakor véletlen *seed*-et kapnak, ezért minden szimuláció más-más eredményt fog hozni.

A *Szimuláció eredménye* ablak *Grafikonok* füle a következőképpen néz ki:



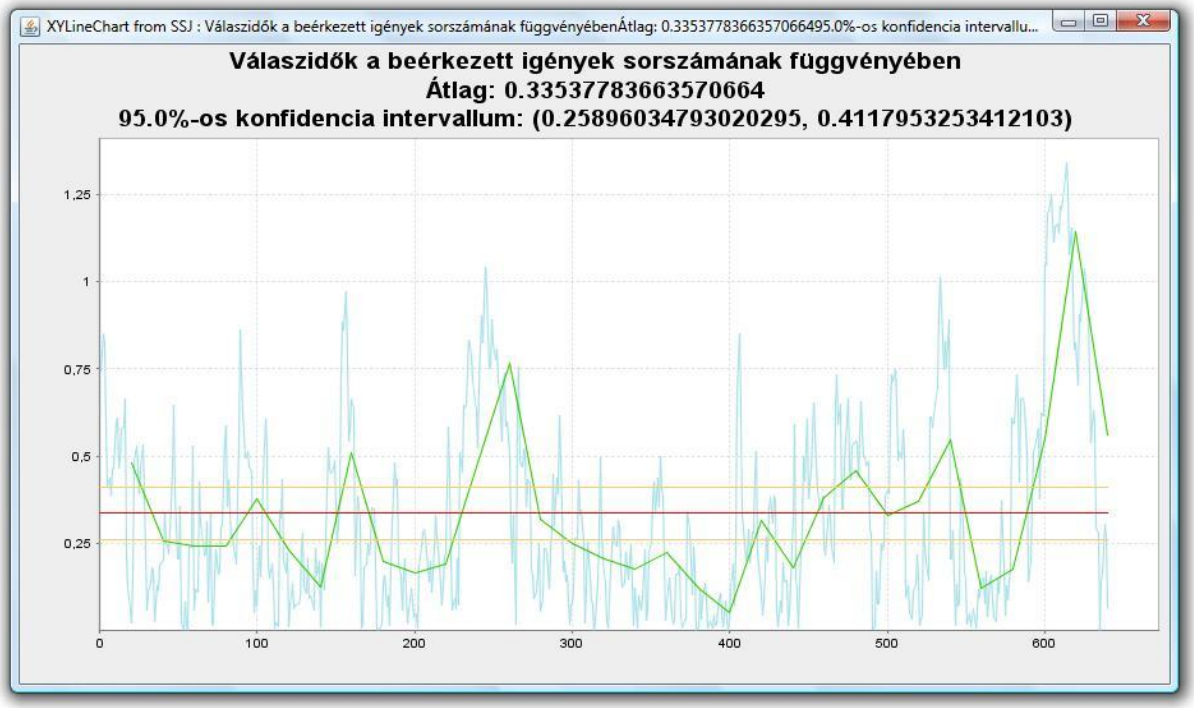
Ábra 7.

A grafikon rajzolásához ki kell választani, hogy melyik mintasorozatból szeretnénk elkészítni a grafikont. Csak akkor válik aktívvá a *Grafikon rajzolása* gomb, ha van kiválasztott rendszerjellemző.

A grafikonon a mért adatok a beérkezett igények függvényében jelennek meg, kivéve a kihasználtságot, ami az idő függvényében jelenik meg. Az ablak alján beállíthatjuk a mintaátlaghoz tartozó konfidencia intervallum meghatározásához szükséges paramétereket, melyek:

- *kötegek mérete*: itt adható meg, hogy a *batch-means method* mekkora hosszúságú intervallumokra bontsa a mintasort. Általában 20-30 szokott lenni ez a méret, ugyanis a kötegeknek elég hosszúaknak kell lenniük, hogy az Y_i-k nagyjából függetlenek legyenek. Ha ezek nem függetlenek, akkor a korreláció pozitív, így a valódi konfidencia intervallum nagyobb, mint, amit a becslés mutatni fog.
- *konfidencia szint*: itt állítható be a konfidencia intervallum konfidencia szintje

Ezek beállítása utána a *Grafikon rajzolása* gomb megpróbálja előállítani a grafikont. Előfordulhat viszont olyan eset, hogy túl kevés esemény történt a szimulációs időben ahhoz, hogy a *batch-means method* konfidencia intervallumot tudjon számolni. Ez többek közt abból adódik, hogy a *batch-means method* számára átadott megfigyelések elejéről az első 20 megfigyelés eltávolításra kerül, annak érdekében, hogy a rendszer már stacionárius állapotban legyen a vizsgálat során. Így tehát legalább 20+*kötegméret* mintára van szükség, hogy a grafikon megjelenjen. (Nyilván ilyen kis esemény szám mellett a konfidencia intervallum becslése nagyon pontatlan.) Amennyiben a grafikon rajzolható, akkor a grafikont ábrázoló ablak a következőképpen néz ki:



Ábra 8.

A címében olvasható a rendszerjellemző, amit ábrázol, annak átlag, és a megadott konfidencia szint melletti konfidencia intervallum. A görbék a következő adatokat jelölik:

- *kék görbe*: a rendszerjellemző statisztikai mintavételező mintái
- *zöld görbe*: a köteg átlagok (*batch-measn*)
- *piros görbe*: a mintaátlag
- *narancs görbék*: a konfidencia intervallum

5. A szimuláció eredményének összehasonlítása a matematikai modell eredményivel

Ebben a fejezetben szeretném összehasonlítani egy sorbanállási rendszer szimuláció által mért tulajdonságainak értékeit a matematikai modellel. Legyen a választott sorbanállási rendszer a legismertebb M/M/1 típusú.

5.1 M/M/1

Az M/M/1 rendszerben az igények λ paraméterű exponenciális eloszlású beérkezési időközökkel érkeznek az egyetlen kiszolgálóhoz, ami μ paraméterű exponenciális eloszlású kiszolgálási időkkel bír. Sorhossz korlát nincsen. A kiszolgálási elv pedig FIFO. Ekkor az M/M/1 matematika modellje az alábbi képletekkel írja le a rendszer jellemzőit:

Annak valószínűsége, hogy egy érkező igény *k* igényt talál a rendszerben:

$$P_k = \rho^k(1-\rho), \qquad \rho = \frac{\lambda}{\mu}$$

Kihasználtság:

$$U = \frac{\lambda}{\mu} = \rho$$

Átlagos sorhossz:

$$L_q = \frac{\rho^2}{(1-\rho)}$$

Átlagos várakozási idő:

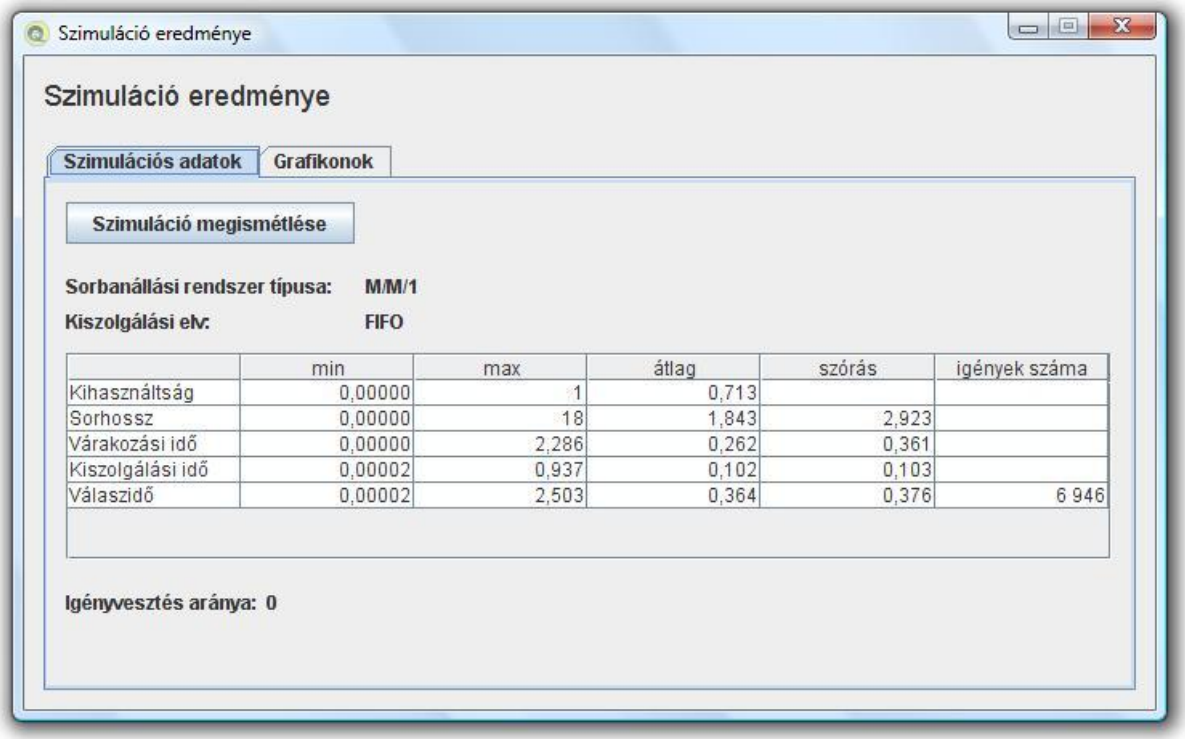
$$\bar{W} = \frac{1}{\mu}\frac{\lambda}{\mu-\lambda}$$

Átlagos válaszidő:

$$\bar{T} = \frac{1}{\mu-\lambda}$$

5.2 A szimuláció eredménye

A 9-es ábra egy olyan M/M/1 sorbanállási rendszer szimulációjának eredményét mutatja, melynek beérkezési időközei $\lambda = 7$ paraméterű exponenciális eloszlásúak, a kiszolgálási idők eloszlása pedig $\mu = 10$ paraméterű. A rendszert 1000 időegységig szimulálta a program.



Ábra 9.

A fenti ábra alapján, és az *5.1*-es pontban leírtak alapján a szimuláció és a matematikai modell eredményeinek összehasonlítása:

	Szimulációs eredmény	*Matematikai modell*
Kihasználtság	0,713	0,7
Átlagos sorhossz	1,843	1.633
Átlagos várakozási idő	0,262	0,233
Átlagos válaszidő	0,364	0,333

Jól látható, hogy a szimulációs eredmények, és a matematikai modell által kapott értékek összhangban vannak, ezzel bizonyítható a program helyessége.

6. Összefoglalás

Dolgozatomban bemutattam egy sorbanállási rendszereket szimuláló program létrehozását, és működési elvét. Törekedtem arra, hogy ne a konkrét implementációt mutassam be, hanem az elvet, ami alapján egy ilyen program létrehozható. Ismertettem az SSJ keretrendszer nyújtotta lehetőségek közül a legfontosabbakat.

Bevezetőmben kiemelten foglalkoztam a téma oktatási vonatkozásával. A bemutatott rendszer megteremti a lehetőségét egy olyan új oktatás tematikának is, ami a sorbanállási rendszereket a szimulációs oldalról közelíti meg. A sorbanállási rendszerek matematikai modellezése mellett létjogosultságát érzem a szimuláció oktatásának is, hiszen bizonyos esetben egyszerűbb megalkotni egy szimulátort, mint egy matematikai modellt.

Az SSJ-ben rengeteg lehetőség rejlik. A szakdolgozat, és a program csak a diszkrét esemény szimuláció alapjait mutatta be. Tekinthető úgy is, hogy egy nagyobb projekt alapjait készítettem el, hiszen léteznek a *Qsim* által támogatott sorbanállási rendszereknél sokkal bonyolultabbak is. Például szimulációval viszonylag egyszerűen konstruálható olyan rendszer, melynek beérkezési időközeinek eloszlása a szimuláció ideje alatt megváltozik. Ezt matematikailag rendkívül bonyolult lenne modellezni. Az SSJ megteremti a lehetőséget a matematikában kevésbé járatos, de a programozásban járatos hallgatók számára, hogy aktívan foglalkozzanak sorbanállási rendszerekkel. Nyilván ezek az empirikus adatok nem olyan meggyőzőek, mint egy bizonyítható matematikai modell, viszont szimulátorok konstruálása sokkal rugalmasabb feladat.

Felvetődhet a kérdés, hogy mi értelem saját szimulátorokat írni, ha a világon számtalan szimulátor program létezik már. Elsősorban azért, mert így semmiféle megkötés nincs egy új rendszer létrehozásánál. Nem kell a szimulátor program korlátaihoz alkalmazkodni. Továbbá az SSJ Java alapú, így a világ egyik legelterjedtebb programozási nyelvének erejét fordíthatjuk a sorbanállási rendszerek vizsgálatára.

7. Irodalomjegyzék

[1] **Sztrik János** (2004) *Bevezetés a sorbanállási elméletbe és alkalmazásaiba*, mobiDIÁK könyvtár

[2] **Solt György** (2006) *Valószínűségszámítás*, Műszaki Kiadó, Budapest

[3] **J. Virtamo** (2005) *Queueing Course, from Finland. Complete lecture notes* (http://www.netlab.hut.fi/opetus/s383143/kalvot/english.shtml)

[4] **John N. Daigle** (2004) *Queueing Theory with Applications to Packet Telecommunication*

[5] **Jerry Banks, John Carson, Barry L. Nelson, David Nico** (2004) *Discrete-Event System Simulation (4th Edition) (Prentice-Hall International Series in Industrial and Systems)*

[6] **Gunter Bolch, Stefan Greiner, Hermann de Meer, Kishor Shridharbhai Trivedi** (2006) *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*

[7] **Gross D., C.M. Harris** (1985) *Fundamentals of Queueing Theory, (2nd Edition)*

[8] **Giovanni Giambene** (2005) *Queuing Theory and Telecommunications: Networks and Applications*

[9] **Nagy Gusztáv**, *Java Swing jegyzet* (http://javaprog.nagygusztav.hu/?q=node/36)

[10] *The Swing Tutorial* (http://java.sun.com/docs/books/tutorial/uiswing/)

[11] **C Chien** – (1994) *Batch size selection for the batch means method - Proceedings of the 26th conference on Winter simulation*

[12] **Natalie M Steiger, Emily K Lada, James R Wilson, Jeffrey A Joines, Christos Alexopoulos, David Goldsman** (2005) *ASAP3: A Batch Means Procedure for Steady-State Simulation Analysis - ACM Transactions on Modeling and Computer Simulation, Vol. 15, No. 1, January 2005, Pages 39–73.*

[13] **P. L'Ecuyer, L. Meliani, J. Vaucher**. (2002) SSJ: *A framework for stochastic simulation in Java. (Proceedings of the 2002 Winter Simulation Conference, pages 234–242. IEEE Press, 2002.)*

[14] **Pierre L'Ecuyer** (2008) *Official SSJ-2.1.1 documentation* (http://www.iro.umontreal.ca/~simardr/ssj/doc/html/index.html

[15] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package charts - Charts generation*, (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guidecharts.pdf)

[16] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package probdist - Probability Distributions*, (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guideprobdist.pdf)

[17] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package randvar - Generating Non-Uniform Random Numbers*, (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guiderandvar.pdf)

[18] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package rng - Random Number Generators*, (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guiderng.pdf)

[19] **Pierre L'Ecuyer** (2008) *SSJ User's Guide – Package simprocs - Process-driven Simulation Management* (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guidesimprocs.pdf)

[20] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package simevents - Simulation Clock and Event List Management*, (http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guidesimevents.pdf)

[21] **Pierre L'Ecuyer** (2008) *SSJ User's Guide - Package stat - Tools for Collecting* (Statistics, http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guidestat.pdf)

8. Mellékletek

1 db CD-ROM, amely a következőket tartalmazza:

- A *Qsim* programot (`Qsim` könyvtár)
- A *Qsim* forráskódját (`forráskód` könyvtár)
- Az SSJ keretrendszer 2.1.1-es verzióját (`ssj` könyvtár)